15-721 ADVANCED DATABASE SYSTEMS

Lecture #20 – Query Compilation

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Background
Code Generation / Transpilation
JIT Compilation (LLVM)
Real-world Implementations

HEKATON REMARK

After switching to an in-memory DBMS, the only way to increase throughput is to reduce the number of instructions executed.

→ To go **10x** faster, the DBMS must execute **90%** fewer instructions…

→ To go **100x** faster, the DBMS must execute **99%** fewer instructions…

COMPILATION IN THE MICROSOFT SQL SERVER HEKATON ENGINE
IEEE Data Engineering Bulletin 2011

OBSERVATION

The only way that we can achieve such a reduction in the number of instructions is through **code specialization**.

This means generating code that is specific to a particular task in the DBMS.

Most code is written to make it easy for humans to understand rather than performance…

EXAMPLE DATABASE

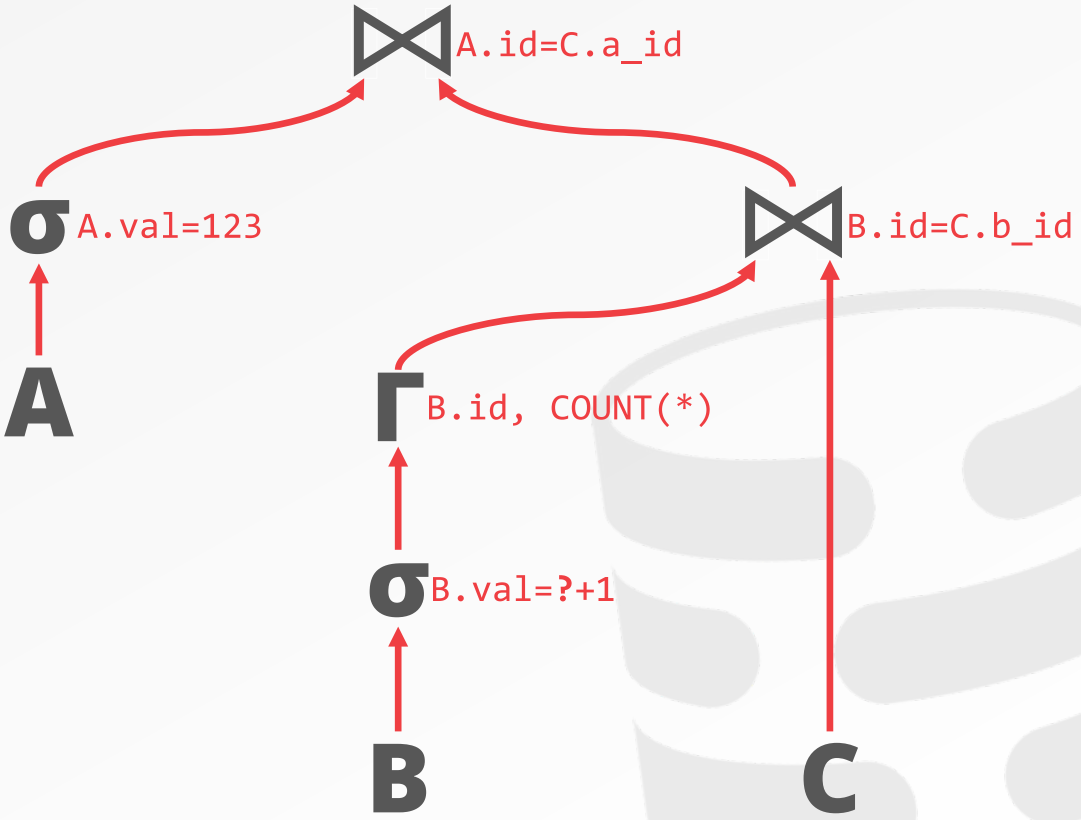
```
CREATE TABLE A (
  id INT PRIMARY KEY,
  val INT
);
```

```
CREATE TABLE B (
  id INT PRIMARY KEY,
  val INT
);
```

```
CREATE TABLE C (
  a_id INT REFERENCES A(id),
  b_id INT REFERENCES B(id),
  PRIMARY KEY (a_id, b_id)
);
```

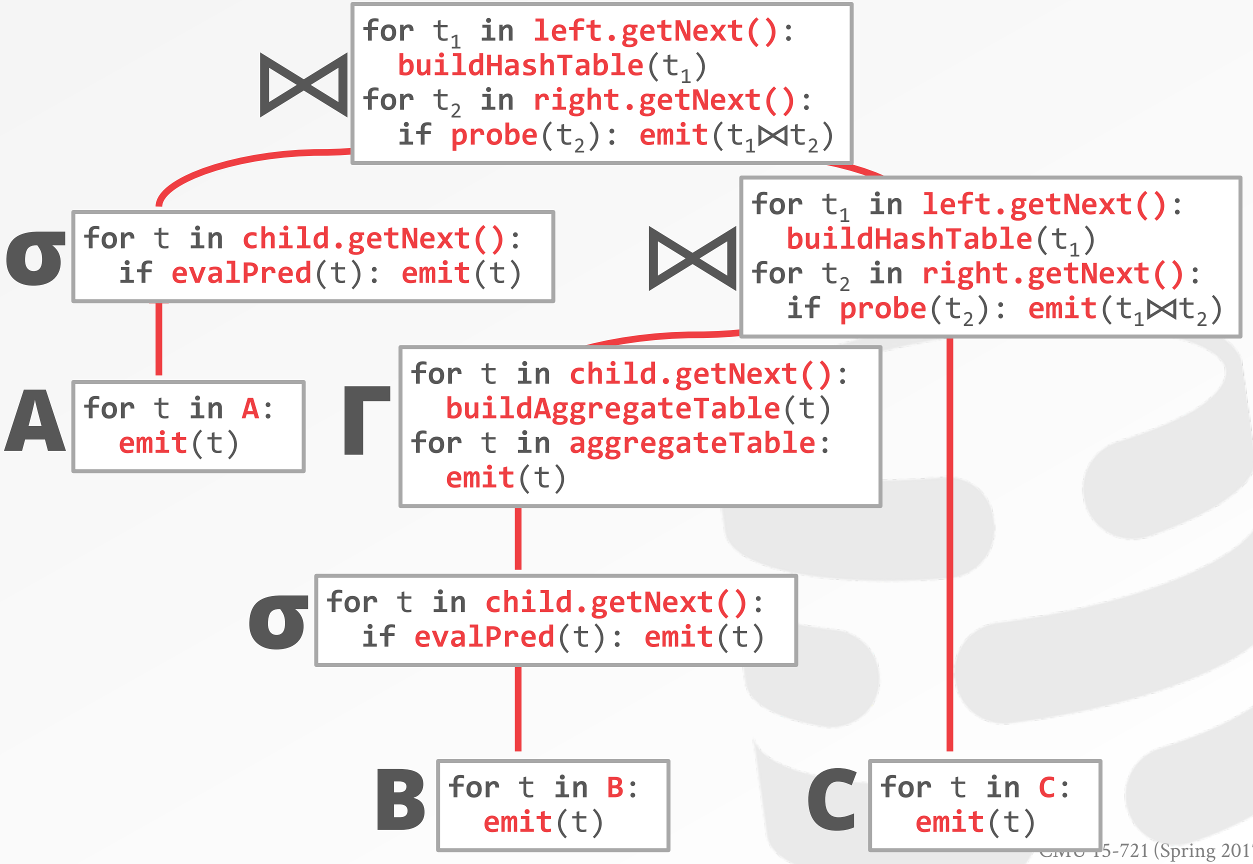
QUERY INTERPRETATION

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



QUERY INTERPRETATION

```sql
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



PREDICATE INTERPRETATION

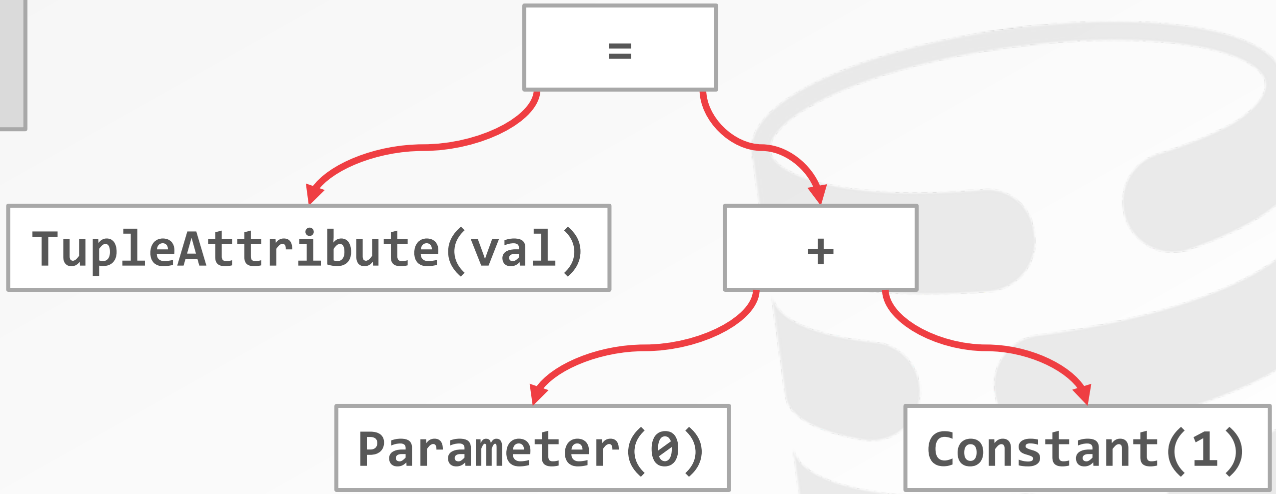
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

PREDICATE INTERPRETATION

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B→(int:id, int:val)

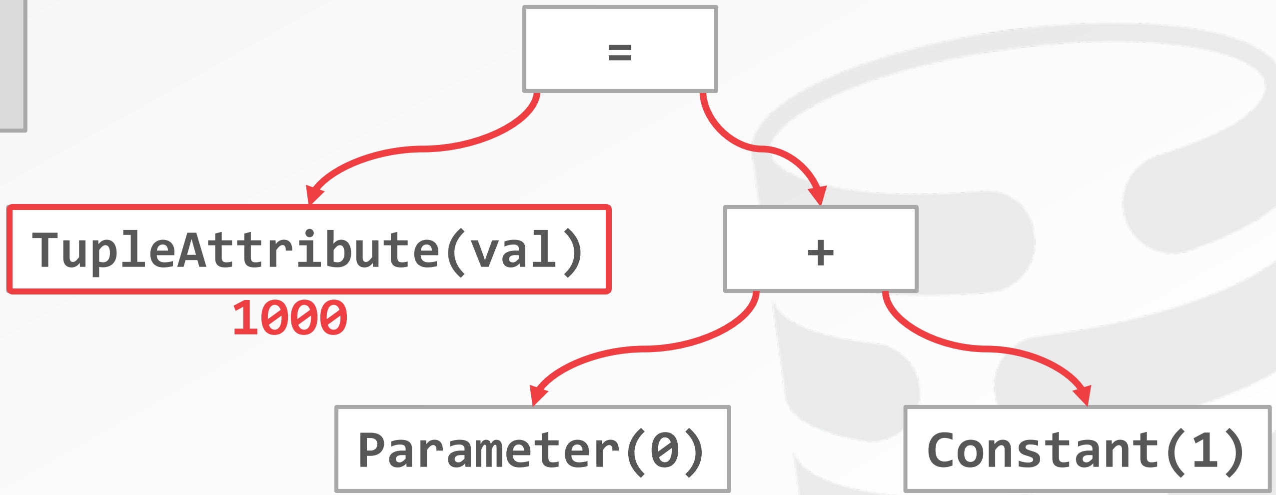


PREDICATE INTERPRETATION

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B→(int:id, int:val)

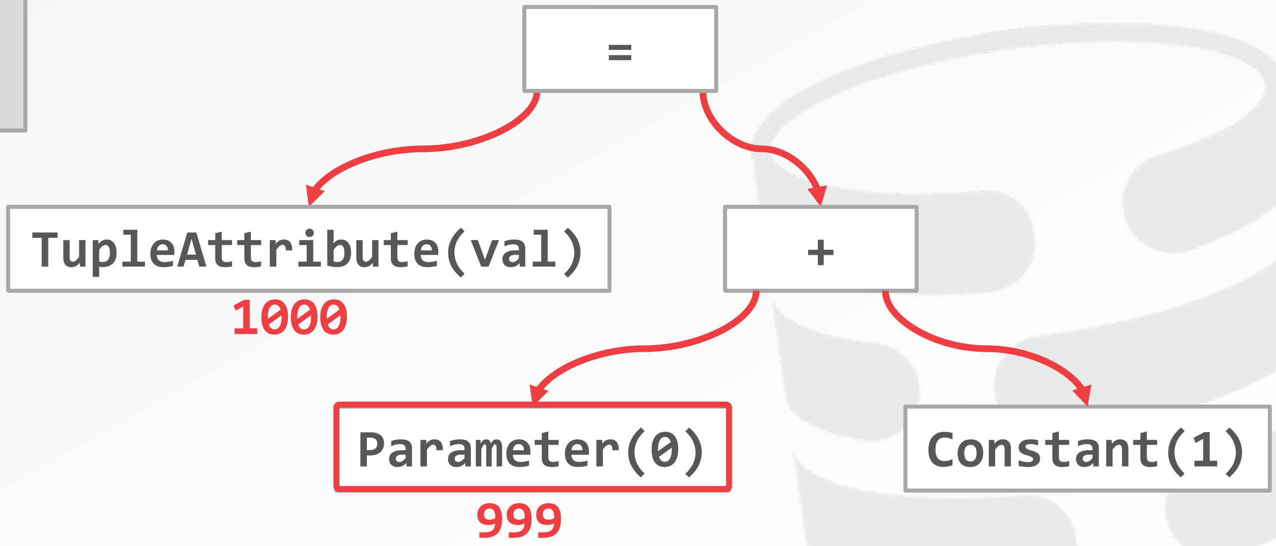


PREDICATE INTERPRETATION

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B→(int:id, int:val)

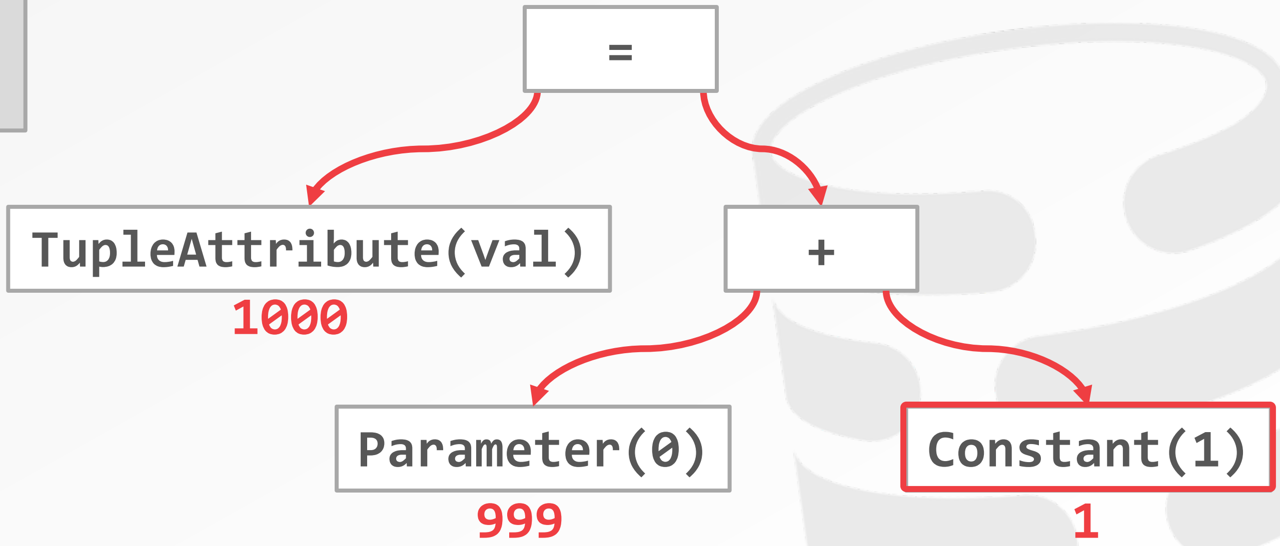


PREDICATE INTERPRETATION

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B→(int:id, int:val)

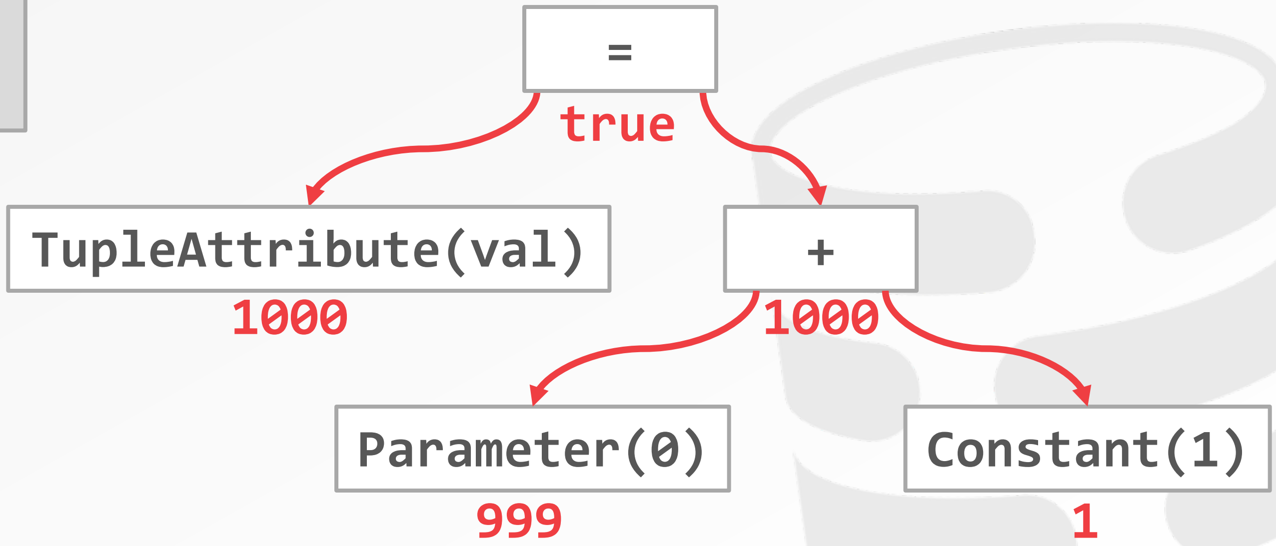


PREDICATE INTERPRETATION

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Execution Context

Current Tuple	Query Parameters	Table Schema
(123, 1000)	(int:999)	B→(int:id, int:val)



CODE SPECIALIZATION

Any CPU intensive entity of database can be natively compiled if they have a similar execution pattern on different inputs.

→ Access Methods
→ Stored Procedures
→ Operator Execution
→ Predicate Evaluation
→ Logging Operations

BENEFITS

Attribute types are known ***a priori***.

→ Data access function calls can be converted to inline pointer casting.

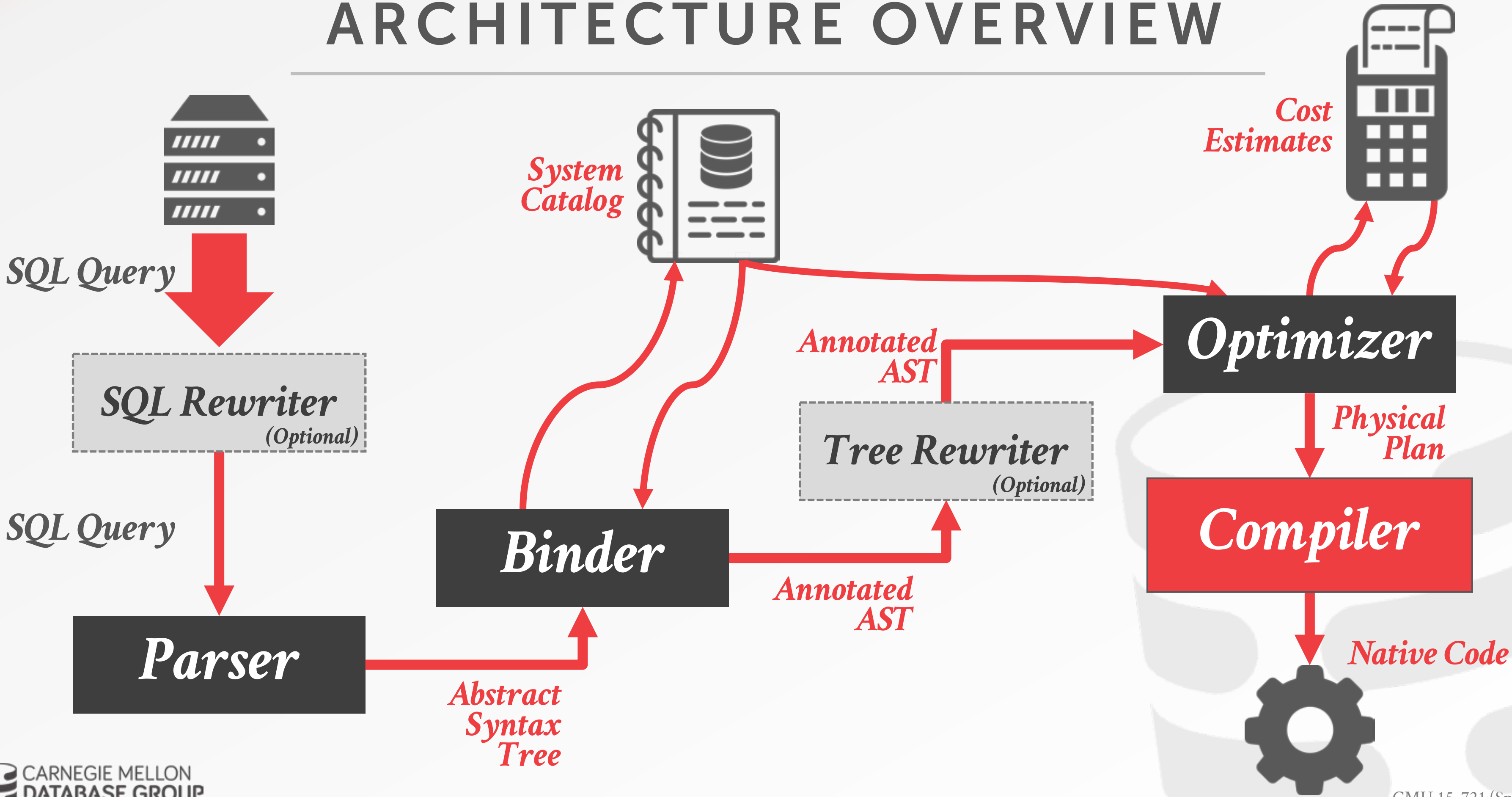
Predicates are known ***a priori***.

→ They can be evaluated using primitive data comparisons.

No function calls in loops

→ Allows the compiler to efficiently distribute data to registers and increase cache reuse.

ARCHITECTURE OVERVIEW



CODE GENERATION

Approach #1: Transpilation

→ Write code that converts a relational query plan into C/C++ and then run it through a conventional compiler to generate native code.

Approach #2: JIT Compilation

→ Generate an ***intermediate representation*** (IR) of the query that can be quickly compiled into native code .

HIQUE – CODE GENERATION

For a given query plan, create a C/C++ program that implements that query's execution.
→ Bake in all the predicates and type conversions.

Use an off-shelf compiler to convert the code into a shared object, link it to the DBMS process, and then invoke the exec function.

GENERATING CODE FOR HOLISTIC QUERY EVALUATION
ICDE 2010

OPERATOR TEMPLATES

```
SELECT * FROM A WHERE A.val = ? + 1
```

OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. ***Get schema in catalog for table.***
2. ***Calculate offset based on tuple size.***
3. ***Return pointer to tuple.***

OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

Templated Plan

```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t ∗ tuple_size
    val = (tuple+predicate_offset) + 1
    if (val == parameter_value):
      emit(tuple)
```

OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

Templated Plan

```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t ∗ tuple_size
    val = (tuple+predicate_offset) + 1
    if (val == parameter_value):
      emit(tuple)
```

OPERATOR TEMPLATES

Interpreted Plan

```
for t in range(table.num_tuples):
  tuple = get_tuple(table, t)
  if eval(predicate, tuple, params):
    emit(tuple)
```

1. *Get schema in catalog for table.*
2. *Calculate offset based on tuple size.*
3. *Return pointer to tuple.*

1. *Traverse predicate tree and pull values up.*
2. *If tuple value, calculate the offset of the target attribute.*
3. *Perform casting as needed for comparison operators.*
4. *Return true / false.*

Templated Plan

```
tuple_size = ###
predicate_offset = ###
parameter_value = ###

for t in range(table.num_tuples):
    tuple = table.data + t ∗ tuple_size
    val = (tuple+predicate_offset) + 1
    if (val == parameter_value):
      emit(tuple)
```

DBMS INTEGRATION

The generated query code can invoke any other function in the DBMS.

This allows it to use all the same components as interpreted queries.

→ Concurrency Control
→ Logging / Checkpoints
→ Indexes

EVALUATION

Generic Iterators
→ Canonical model with generic predicate evaluation.

Optimized Iterators
→ Type-specific iterators with inline predicates.

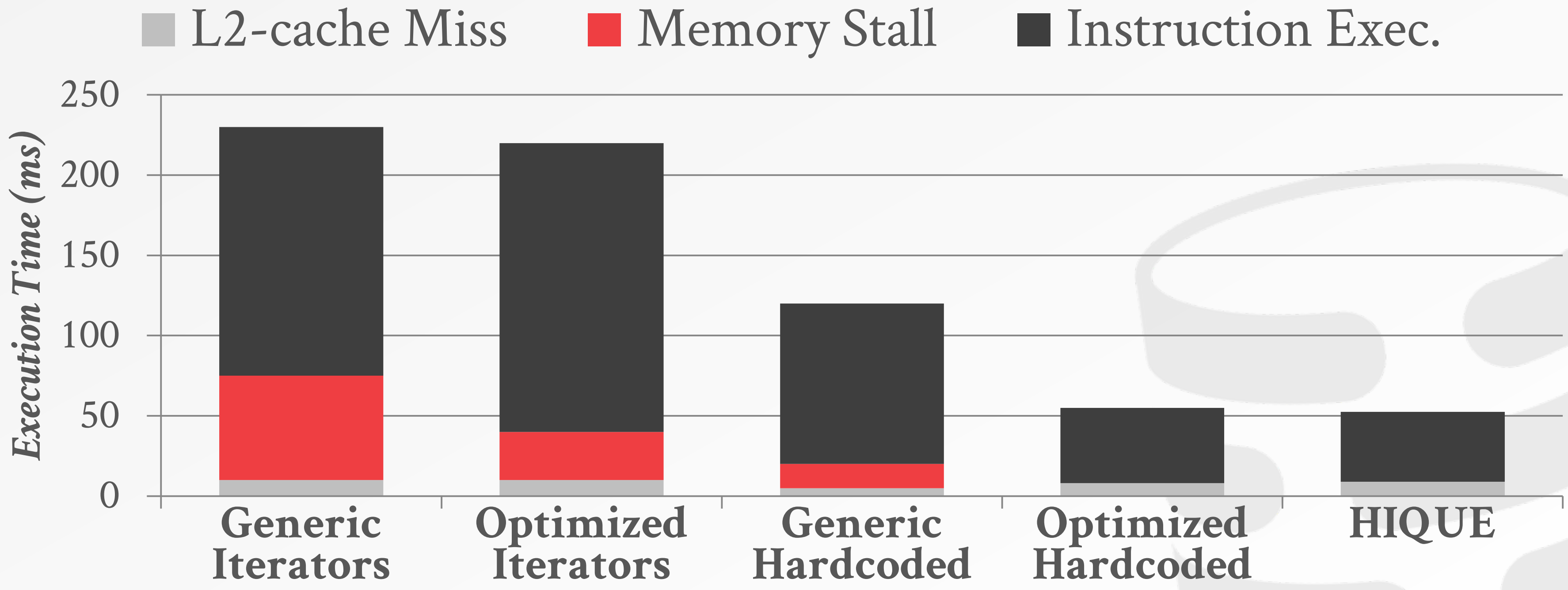
Generic Hardcoded
→ Handwritten code with generic iterators/predicates.

Optimized Hardcoded
→ Direct tuple access with pointer arithmetic.

HIQUE
→ Query-specific specialized code.

QUERY COMPILATION EVALUATION

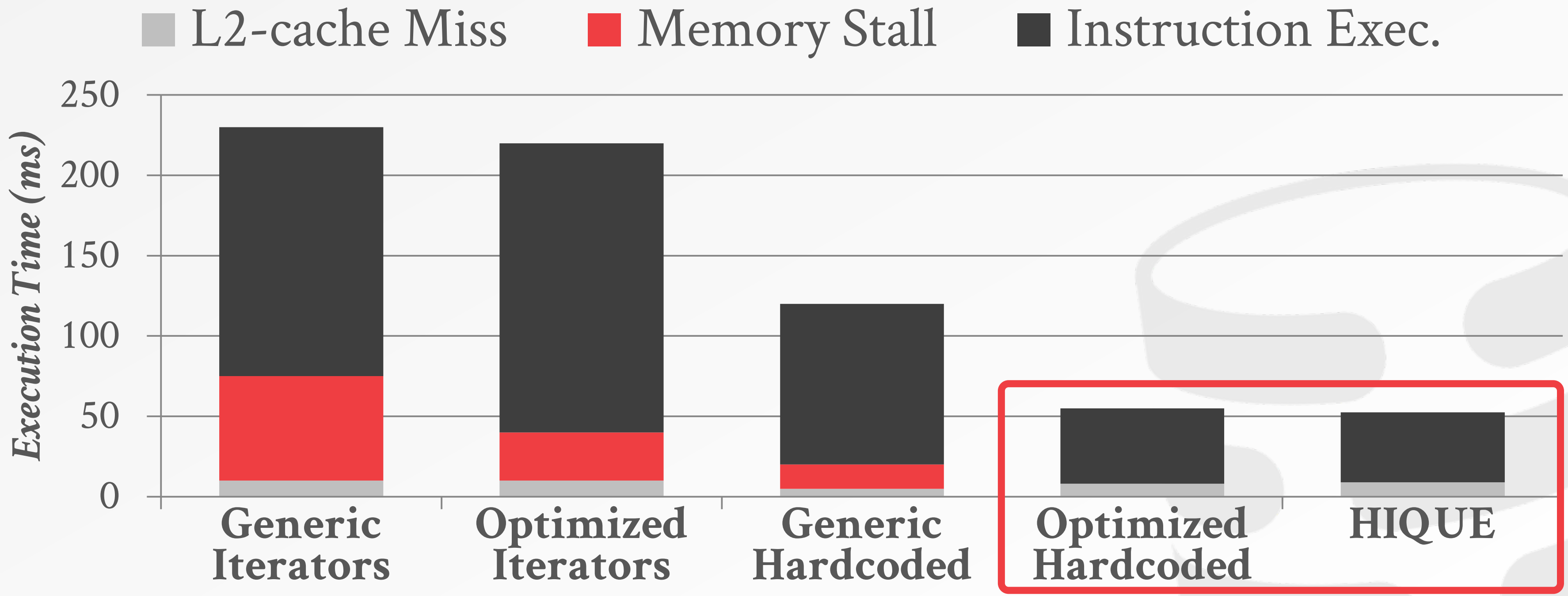
Intel Core 2 Duo 6300 @ 1.86GHz
Join Query: 10k⋈10k→10m



Source: Konstantinos Krikellas

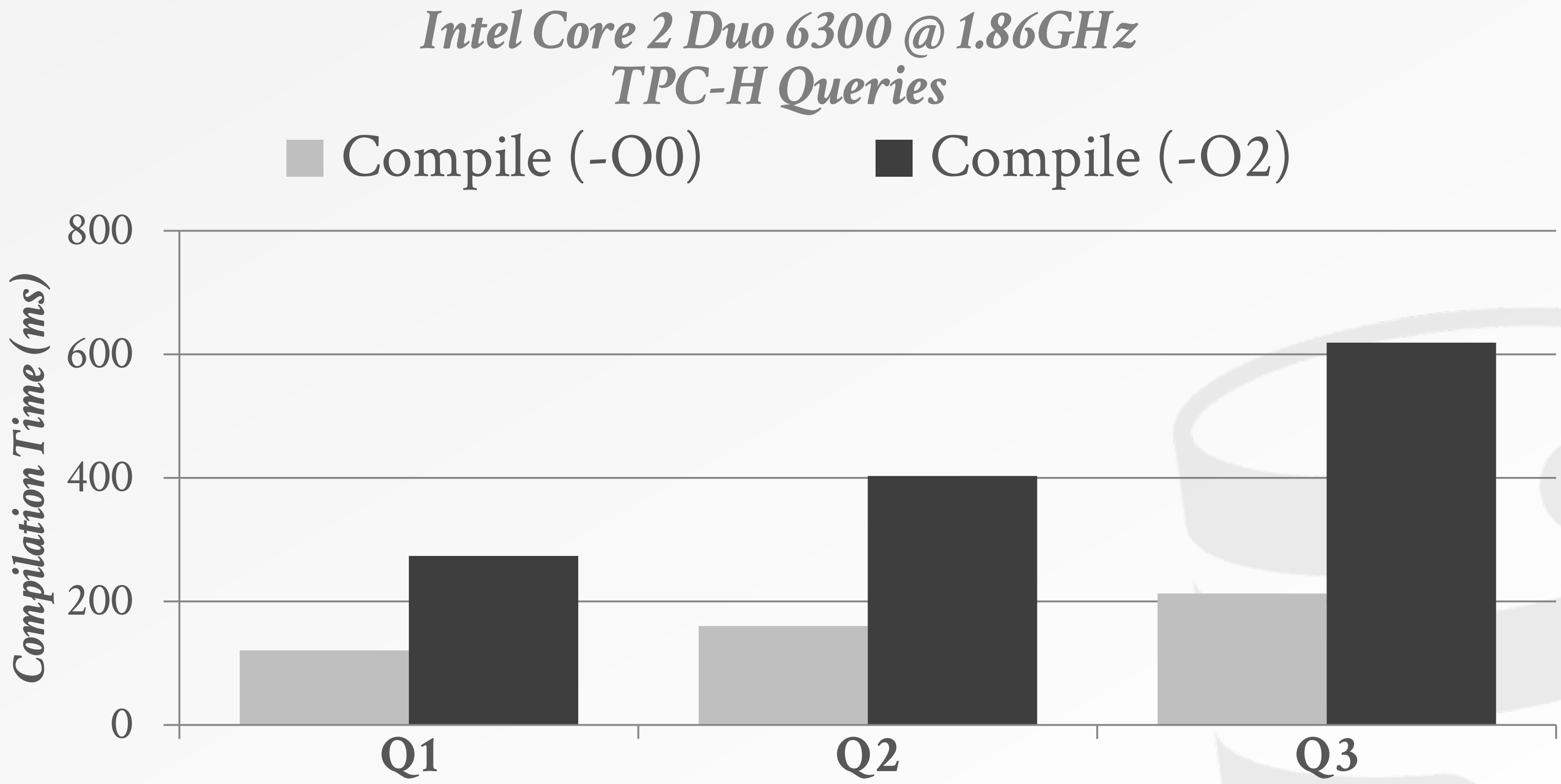
QUERY COMPILATION EVALUATION

Intel Core 2 Duo 6300 @ 1.86GHz
Join Query: 10k⋈10k→10m



Source: Konstantinos Krikellas

QUERY COMPILATION COST



Source: Konstantinos Krikellas

OBSERVATION

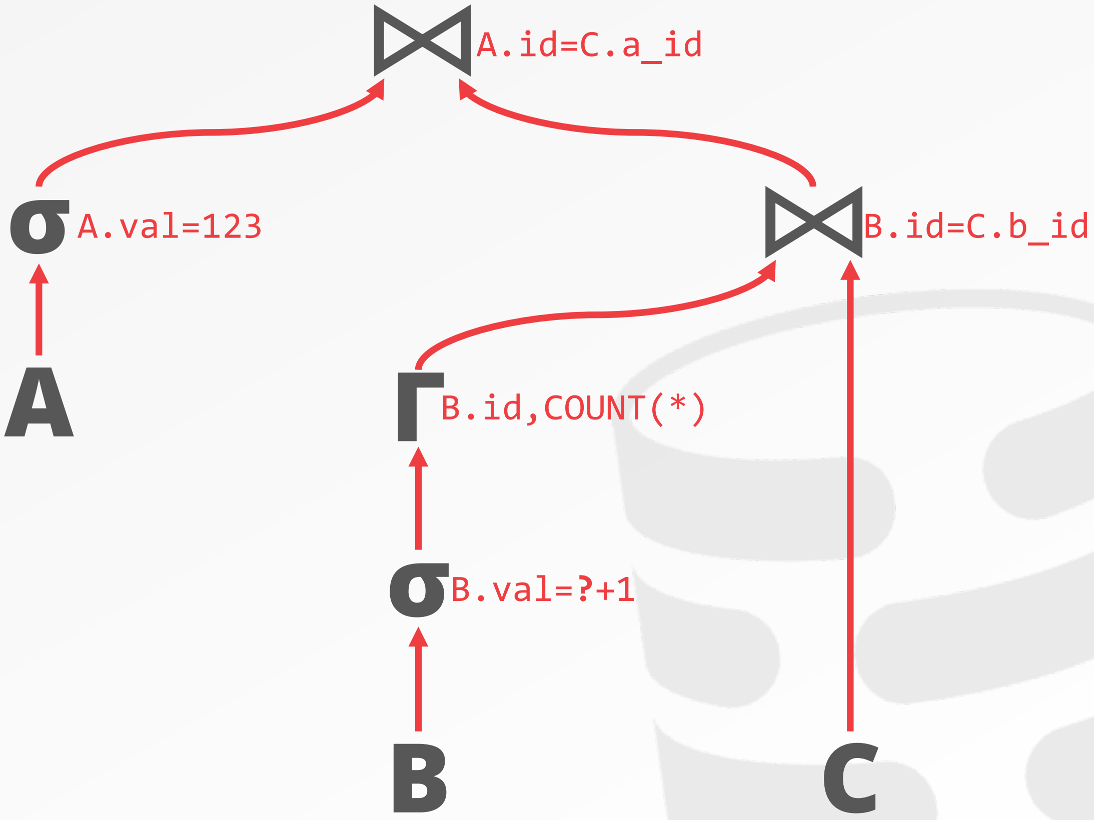
Relational operators are a useful way to reason about a query but are not the most efficient way to execute it.

It takes a (relatively) long time to compile a C/C++ source file into executable code.

HIQUE does not allow for full pipelining…

PIPELINED OPERATORS

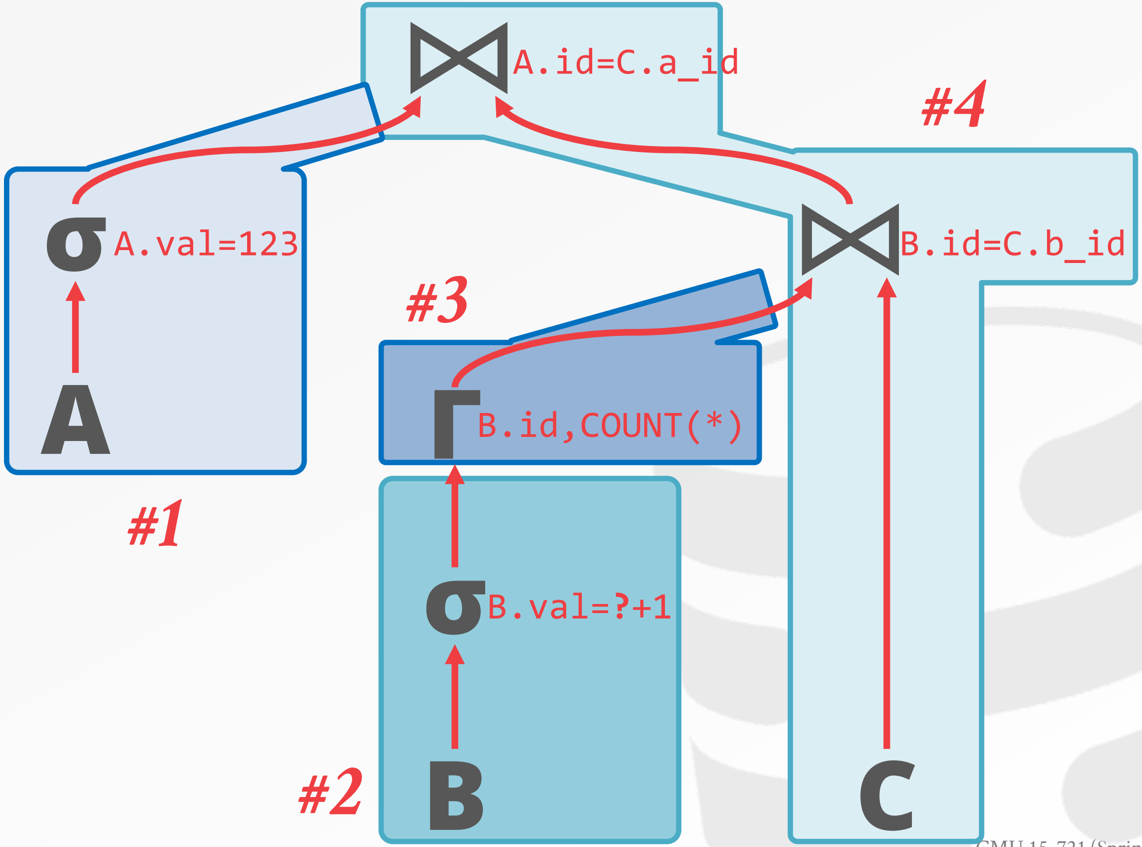
```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```



PIPELINED OPERATORS

```
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
  WHERE A.val = 123
    AND A.id = C.a_id
    AND B.id = C.b_id
```

Pipeline Boundaries



HYPER – JIT QUERY COMPILATION

Compile queries in-memory into native code using the LLVM toolkit.

Organizes query processing in a way to keep a tuple in CPU registers for as long as possible.
→ Push-based vs. Pull-based
→ Data Centric vs. Operator Centric

EFFICIENTLY COMPILING EFFICIENT QUERY PLANS FOR MODERN HARDWARE
VLDB 2011

LLVM

Collection of modular and reusable compiler and toolchain technologies.

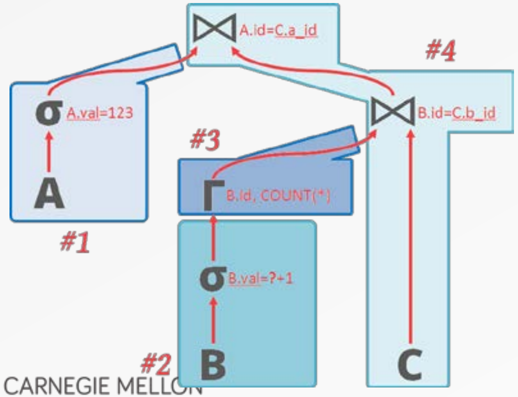
Core component is a low-level programming language (IR) that is similar to assembly.

Not all of the DBMS components need to be written in LLVM IR.

→ LLVM code can make calls to C++ code.

PUSH-BASED EXECUTION

```sql
SELECT *
  FROM A, C,
   (SELECT B.id, COUNT(*)
      FROM B
     WHERE B.val = ? + 1
     GROUP BY B.id) AS B
 WHERE A.val = 123
   AND A.id = C.a_id
   AND B.id = C.b_id
```



Generated Query Plan

#1
```
for t in A:
  if t.val == 123:
    Materialize t in HashTable ⋈(A.id=C.a_id)
```

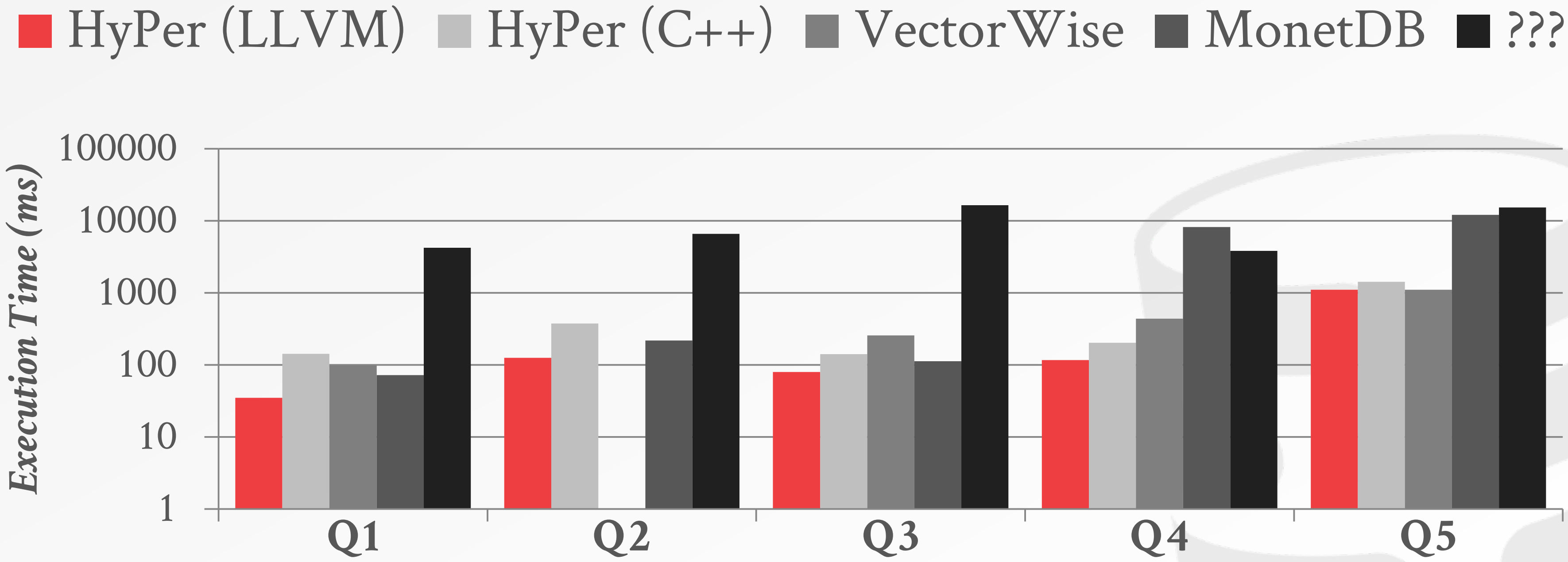
#2
```
for t in B:
  if t.val == <param> + 1:
    Aggregate t in HashTable Γ(B.id)
```

#3
```
for t in Γ(B.id):
  Materialize t in HashTable ⋈(B.id=C.b_id)
```

#4
```
for t3 in C:
  for t2 in ⋈(B.id=C.b_id):
    for t1 in ⋈(A.id=C.a_id):
      emit(t1⋈t2⋈t3)
```

QUERY COMPILATION EVALUATION

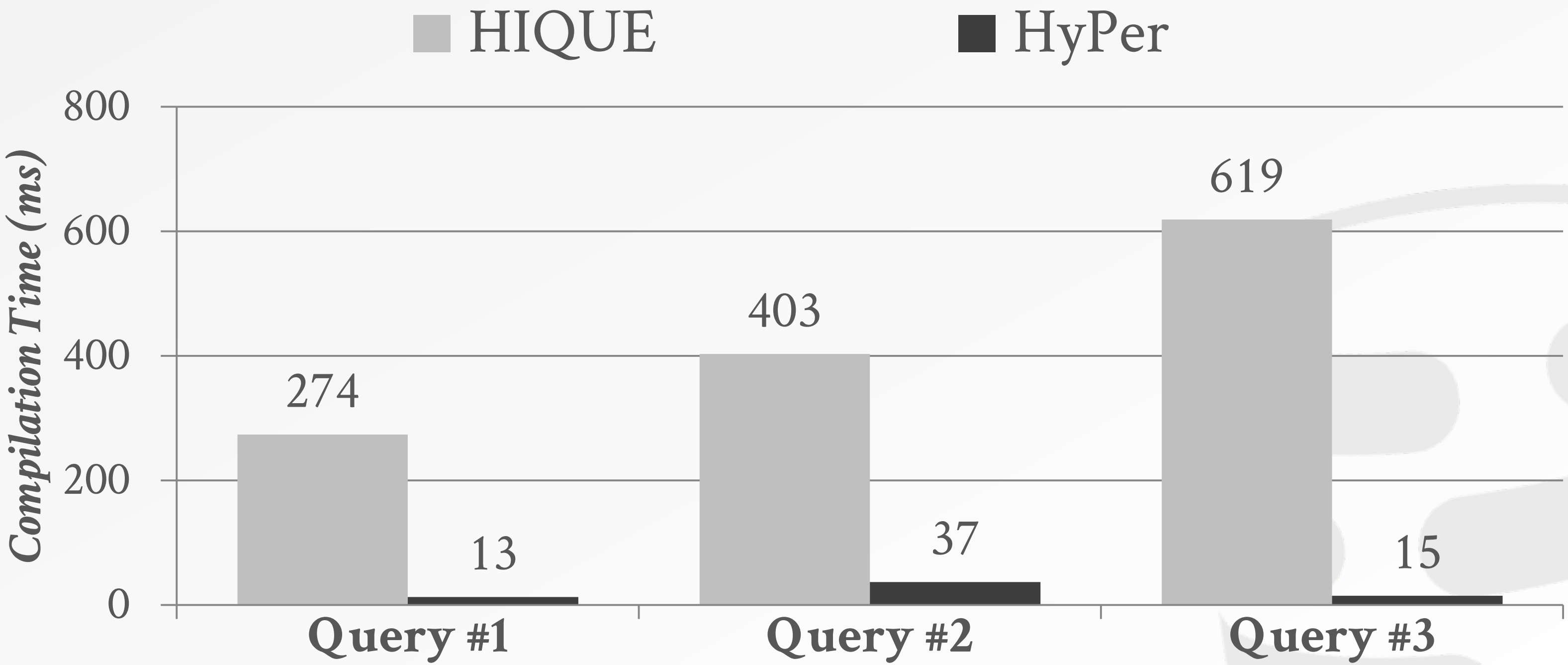
Dual Socket Intel Xeon X5770 @ 2.93GHz
TPC-H Queries



Source: Thomas Neumann

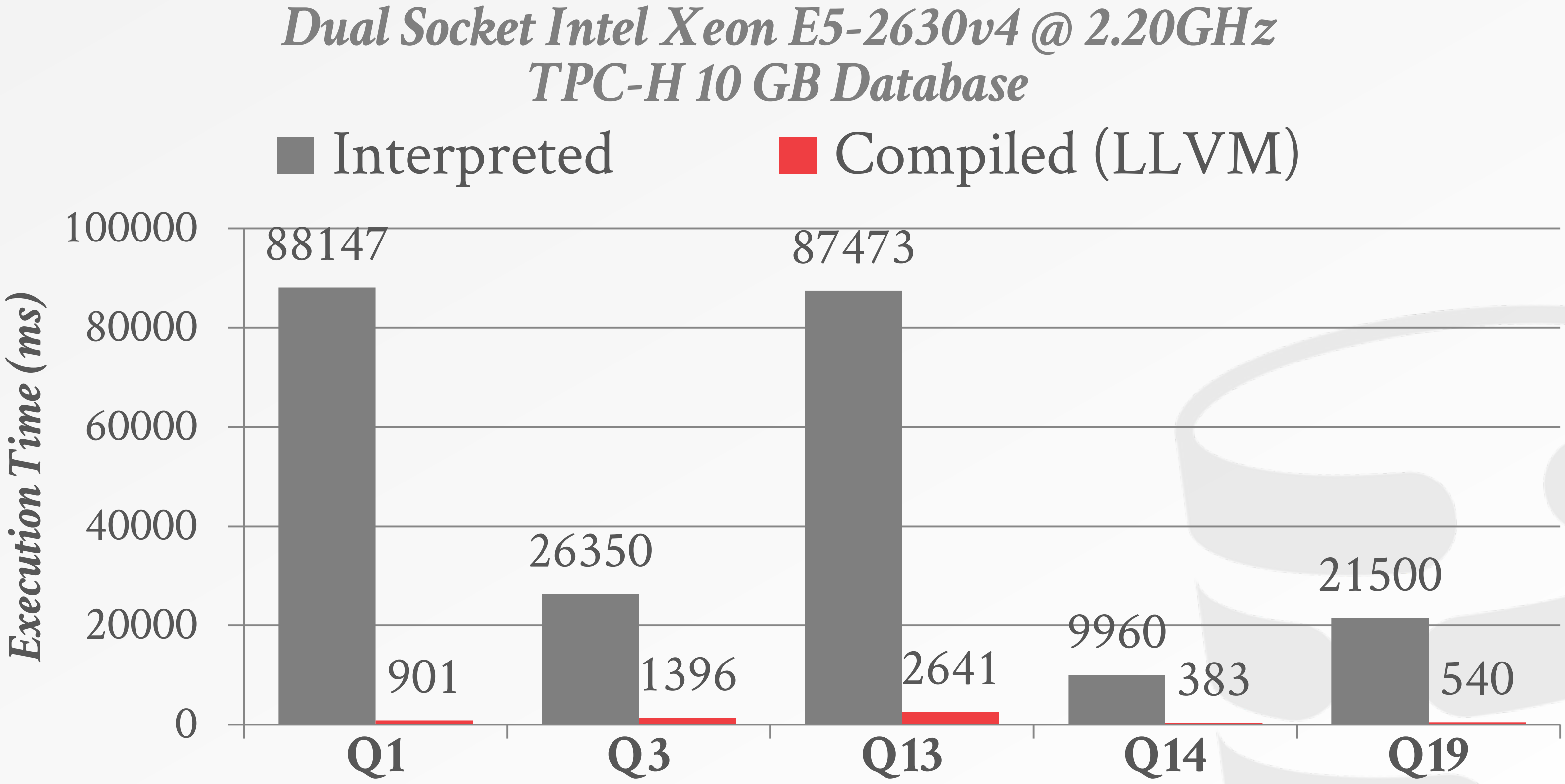
QUERY COMPILATION COST

HIQUE (-O2) vs. HyPer
TPC-H Queries



Source: Konstantinos Krikellas

NEXT-GEN PELOTON



Source: Prashanth Menon

REAL-WORLD IMPLEMENTATIONS

IBM System R

Oracle

Microsoft Hekaton

Cloudera Impala

Actian Vector (*formerly Vectorwise*)

MemSQL

VitesseDB

IBM SYSTEM R

A primitive form of code generation and query compilation was used by IBM in 1970s.

→ Compiled SQL statements into assembly code by selecting code templates for each operator.

Technique was abandoned when IBM built DB2:

→ High cost of external function calls

→ Poor portability

→ Software engineer complications

A HISTORY AND EVALUATION OF SYSTEM R
Communications of the ACM 1981

ORACLE

Convert PL/SQL stored procedures into Pro*C code and then compiled into native C/C++ code.

They also put Oracle-specific operations **directly** in the SPARC chips as co-processors.

→ Memory Scans
→ Bit-pattern Dictionary Compression
→ Vectorized instructions designed for DBMSs
→ Security/encryption

MICROSOFT HEKATON

Can compile both procedures and SQL.

→ Non-Hekaton queries can access Hekaton tables through compiled inter-operators.

Generates C code from an imperative syntax tree, compiles it into DLL, and links at runtime.

Employs safety measures to prevent somebody from injecting malicious code in a query.

COMPILATION IN THE MICROSOFT SQL SERVER HEKATON ENGINE
IEEE Data Engineering Bulletin 2011

CLOUDERA IMPALA

LLVM JIT compilation for predicate evaluation and record parsing.

→ Not sure if they are also doing operator compilation.

Optimized record parsing is important for Impala because they need to handle multiple data formats stored on HDFS.

IMPALA: A MODERN, OPEN-SOURCE SQL ENGINE FOR HADOOP
CIDR 2015

ACTIAN VECTOR

Pre-compiles thousands of "primitives" that perform basic operations on typed data.

→ Example: Generate a vector of tuple ids by applying a less than operator on some column of a particular type.

The DBMS then executes a query plan that invokes these primitives at runtime.

→ Function calls are amortized over multiple tuples

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

ACTIAN VECTOR

```
size_t scan_lessthan_int32(int *res, int32_t *col, int32_t val) {
  size_t k = 0;
  for (size_t i = 0; i < n; i++)
    if (col[i] < val) res[k++] = i;
  return (k);
}
```

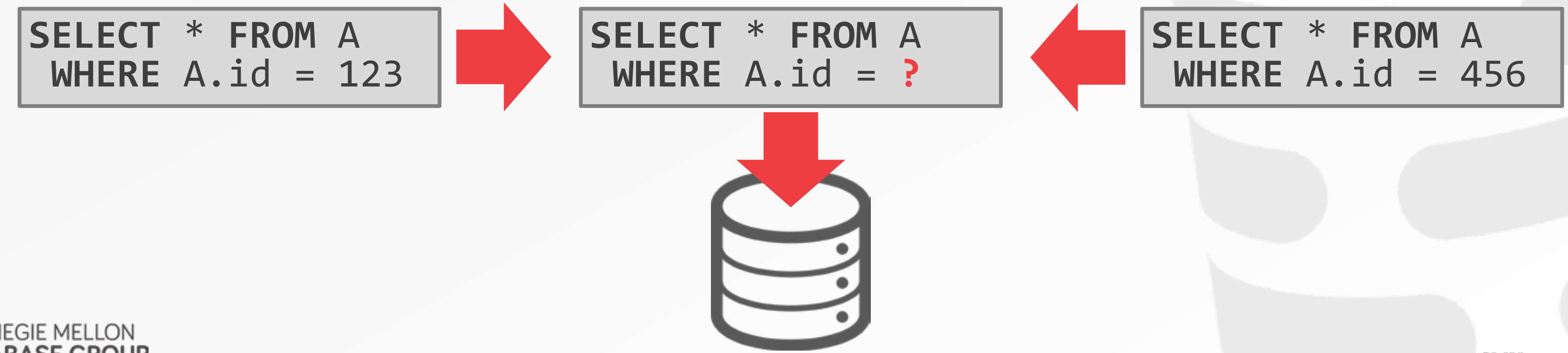
```
size_t scan_lessthan_double(int *res, int32_t *col, double val) {
  size_t k = 0;
  for (size_t i = 0; i < n; i++)
    if (col[i] < val) res[k++] = i;
  return (k);
}
```

MICRO ADAPTIVITY IN VECTORWISE
SIGMOD 2013

MEMSQL (PRE–2016)

Performs the same C/C++ code generation as HIQUE and then invokes gcc.

Converts all queries into a parameterized form and caches the compiled query plan.



MEMSQL (2016–PRESENT)

A query plan is converted into an imperative plan expressed in a high-level imperative DSL.

→ MemSQL Programming Language (MPL)
→ Think of this as a C++ dialect.

The DSL then gets converted into a second language of opcodes.

→ MemSQL Bit Code (MBC)
→ Think of this as JVM byte code.

Finally the DBMS compiles the opcodes into LLVM IR and then to native code.

Source: Drew Paroski

VITESSEDB

Query accelerator for Postgres/Greenplum that uses LLVM + intra-query parallelism.

→ JIT predicates
→ Push-based processing model
→ Indirect calls become direct or inlined.
→ Leverages hardware for overflow detection.

Does not support all of Postgres' types and functionalities. All DML operations are still interpreted.

Source: CK Tan

PARTING THOUGHTS

Query compilation makes a difference but is non-trivial to implement.

The 2016 version of MemSQL is the best query compilation implementation out there.
Hekaton is very good too.

Any new DBMS that wants to compete has to implement query compilation.

NEXT CLASS

Vectorization